Silk Performer 20.5

Java Explorer Help

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
http://www.microfocus.com

2019-10-28

Contents

Tools and Samples

Explains the tools, sample applications and test projects that Silk Performer provides for testing Java and .NET.

Introduction

This introduction serves as a high-level overview of the different test approaches and tools, including Java Explorer, Java Framework, .NET Explorer, and .NET Framework, that are offered by Silk Performer Service Oriented Architecture (SOA) Edition.

Silk Performer SOA Edition Licensing

Each Silk Performer installation offers the functionality required to test .NET and Java components. Access to Java and .NET component testing functionality is however only enabled through Silk Performer licensing options. A Silk Performer SOA Edition license is required to enable access to component testing functionality. Users may or may not additionally have a full Silk Performer license.

What You Can Test With Silk Performer SOA Edition

With Silk Performer SOA Edition you can thoroughly test various remote component models, including:

- Web Services
- .NET Remoting Objects
- Enterprise JavaBeans (EJB)
- Java RMI Objects
- General GUI-less Java and .NET components

Unlike standard unit testing tools, which can only evaluate the functionality of a remote component when a single user accesses it, Silk Performer SOA Edition can test components under concurrent access by up to five virtual users, thereby emulating realistic server conditions. With a full Silk Performer license, the number of virtual users can be scaled even higher. In addition to testing the functionality of remote components, Silk Performer SOA Edition also verifies the performance and interoperability of components.

Silk Performer SOA Edition assists you in automating your remote components by:

- Facilitating the development of test drivers for your remote components
- Supporting the automated execution of test drivers under various conditions, including functional test scenarios and concurrency test scenarios
- Delivering quality and performance measures for tested components

Silk Performer offers the following approaches to creating test clients for remote components:

- Visually, without programming, through Java Explorer and .NET Explorer
- Using an IDE (Microsoft Visual Studio)
- Writing Java code
- Recording an existing client
- Importing JUnit or NUnit testing frameworks
- Importing Java classes
- Importing .NET classes

Provided Tools

Offers an overview of each of the tools provided with Silk Performer for testing Java and .NET.

Silk Performer .NET Explorer

Silk Performer .NET Explorer, which was developed using .NET, enables you to test Web Services, .NET Remoting objects, and other GUI-less .NET objects. .NET Explorer allows you to define and execute complete test scenarios with different test cases without requiring manual programming; everything is done visually through point and click operations. Test scripts are visual and easy to understand, even for staff members who are not familiar with .NET programming languages.

Test scenarios created with .NET Explorer can be exported to Silk Performer for immediate reuse in concurrency and load testing, and to Microsoft Visual Studio for further customization.

Silk Performer Visual Studio Extension

The Silk Performer Visual Studio extension allows you to implement test drivers in Microsoft Visual Studio that are compatible with Silk Performer. Such test drivers can be augmented with Silk Performer features that facilitate test organization, verification, performance measurement, test data generation, and reporting.

Tests created with the extension can be run either within Microsoft Visual Studio, with full access to Silk Performer's functionality, or within Silk Performer, for concurrency and load testing scenarios.

The extension offers the following features:

- Writing test code in any of the main .NET languages (C# or VB.NET).
- Testing Web services / .NET Remoting objects and redirecting HTTP traffic over the Silk Performer Web engine to take advantage of features such as modem simulation and IP-address multiplexing. SOAP envelopes can also be explored using TrueLog Explorer.
- Defining virtual users and their transactions through .NET custom attributes. A BDL script is generated automatically based on the custom attributes that have been applied to classes/methods.
- Running TryScript tests from within Microsoft Visual Studio with the ability to watch virtual user output in a tool window within Microsoft Visual Studio.
- Exploring the results of Try Scripts.

.NET Resources

- *http://msdn.microsoft.com/net*

Silk Performer Java Explorer

Silk Performer Java Explorer, which was developed using Java, enables you to test Web Services, Enterprise JavaBeans (EJB), RMI objects, and other GUI-less Java objects. Java Explorer allows you to define and execute complete test scenarios with multiple test cases without requiring manual programming. Everything can be done visually via point and click operations. Test scripts are visual and easy to understand, even for personnel who are not familiar with Java programming.

Test scenarios created with Java Explorer can be exported to Silk Performer for immediate reuse in concurrency and load testing.

Note: Java Explorer is only compatible with JDK versions 1.2 and later (v1.4 or later recommended).

Java Resources

- *http://java.sun.com*
- *http://www.javaworld.com*

Silk Performer Workbench

Remote component tests that are developed and executed using Java Explorer or .NET Explorer can be executed within Silk Performer Workbench. Silk Performer is an integrated test environment that serves as a central console for creating, executing, controlling and analyzing complex testing scenarios. Java Explorer and .NET Explorer visual test scripts can be exported to Silk Performer by creating Silk Performer Java Framework and .NET Framework projects. While Java Explorer and .NET Explorer serve as test-beds for functional test scenarios, Silk Performer can be used to run the same test scripts in more complex scenarios for concurrency and load testing.

In the same way that Silk Performer is integrated with Java Explorer and .NET Explorer, Silk Performer is also integrated with Silk Performer's Visual Studio .NET Add-On. Test clients created in Microsoft Visual Studio using Silk Performer's Visual Studio .NET Add-On functionality can easily be exported to Silk Performer for concurrency and load testing.

Note: Because there is such a variety of Java development tools available, a Java tool plug-in is not feasible. Instead, Silk Performer offers features that assist Java developers, such as syntax highlighting for Java and the ability to run the Java complier from Silk Performer Workbench.

In addition to the integration of Silk Performer with .NET Explorer, Java Explorer, and Microsoft Visual Studio, you can use Silk Performer to write custom Java and .NET based test clients using Silk Performer's powerful Java and .NET Framework integrations.

The tight integration of Java and .NET as scripting environments for Silk Performer test clients allows you to reuse existing unit tests developed with JUnit and NUnit by embedding them into Silk Performer's framework architecture. To begin, launch Silk Performer and create a new Java or .NET Framework-based project.

In addition to creating test clients visually and manually, Silk Performer also allows you to create test clients by recording the interactions of existing clients, or by importing JUnit test frameworks or existing Java/.NET classes. A recorded test client precisely mimics the interactions of a real client.

Note: The recording of test clients is only supported for Web Services clients.

To create a Web Service test client based on the recording of an existing Web Service client, launch Silk Performer and create a new project of application type `Web Services/XML/SOAP`.

Sample Applications for testing Java and .NET

The sample applications provided with Silk Performer enable you to experiment with Silk Performer's component-testing functionality.

Sample applications for the following component models are provided:

- Web Services
- .NET Remoting
- Java RMI

Public Web Services

Several Web Services are hosted on publicly accessible demonstration servers:

- *http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx*

- *http://demo.borland.com/OrderWebServiceEx/OrderService.asmx*
- *http://demo.borland.com/OrderWebService/OrderService.asmx*
- *http://demo.borland.com/AspNetDataTypes/DataTypes.asmx*

Note: *OrderWebService* provides the same functionality as *OrderWebServiceEx*, however it makes use of SOAP headers in transporting session information, which is not recommended as a starting point for Java Explorer.

.NET Message Sample

The .NET Message Sample provides a .NET sample application that utilizes various .NET technologies:

- Web Services
- ASP.NET applications communicating with Web Services
- WinForms applications communicating with Web Services and directly with .NET Remoting objects.

To access the .NET Message Sample:

If you have Silk Performer SOA Edition: Go to **Start** > **Programs** > **Silk** > **Silk Performer SOA Edition 20.5** > **Sample Applications** > **.NET Framework Samples** .

If you have Silk Performer Enterprise Edition: Go to **Start** > **Programs** > **Silk** > **Silk Performer 20.5** > **Sample Applications** > **.NET Framework Samples** .

.NET Explorer Remoting Sample

The .NET Remoting sample application can be used in .NET Explorer for the testing of .NET Remoting.

To access the .NET Explorer Remoting Sample:

If you have Silk Performer SOA Edition: Go to **Start** > **Programs** > **Silk** > **Silk Performer SOA Edition 20.5** > **Sample Applications** > **.NET Explorer Samples** > **.NET Explorer Remoting Sample** .

If you have Silk Performer Enterprise Edition: Go to **Start** > **Programs** > **Silk** > **Silk Performer 20.5** > **Sample Applications** > **.NET Explorer Samples** > **.NET Explorer Remoting Sample** .

DLL reference for .NET Explorer: `<public user documents>\Silk Performer 20.5\SampleApps\DOTNET\RemotingSamples\RemotingLib\bin\debug\RemotingLib.dll`.

Java RMI Samples

Two Java RMI sample applications are included:

- A simple RMI sample application that is used in conjunction with the sample Java Framework project (`<public user documents>\Silk Performer 20.5\Samples\JavaFramework\RMI`).

  To start the ServiceHello RMI Server, go to: **Start** > **Programs** > **Silk** > **Silk Performer 20.5** > **Sample Applications** > **Java Samples** > **RMI Sample - SayHello**.
- A more complex RMI sample that uses RMI over IIOP is also available. For details on setting up this sample, go to: **Start** > **Programs** > **Silk** > **Silk Performer 20.5** > **Sample Applications** > **Java Samples** > **Product Manager**. This sample can be used with the sample test project that is available at `<public user documents>\Silk Performer 20.5\SampleApps\RMILdap`.

Java RMI can be achieved using two different protocols, both of which are supported by Java Explorer:

- Java Remote Method Protocol (JRMP)
- RMI over IIOP

Java Remote Method Protocol (JRMP)

A simple example server can be found at `<public user documents>\Silk Performer 20.5\SampleApps\Java`.

Launch the batch file `LaunchRemoteServer.cmd` to start the sample server. Then use the Java Explorer **Start Here Wizard** to begin testing RMI objects. Select **RMI** and click **Next**.

The next dialog asks for the RMI registry settings and a classpath where the RMI interfaces for the client can be found. Here are the settings to be used for this example:

Host: `localhost`

Port: `1099`

Client Stub Class: `<public user documents>\Silk Performer 20.5\SampleApps\Java\Lib\sampleRmi.jar`.

RMI over IIOP

A simple example server can be found at: `<public user documents>\Silk Performer 20.5\SampleApps\Java`.

Launch the batch file `LaunchRemoteServerRmiOverIiop.cmd` to start the sample server.

Use the Java Explorer **Start Here Wizard** to begin testing RMI objects. Select `Enterprise JavaBeans/RMI over IIOP` and click **Next**.

The next step asks for the JNDI settings and a classpath where the RMI interfaces for the client can be found. Here are the settings to be provided for this example:

Server: `Sun J2EE Server`

Factory: `com.sun.jndi.cosnaming.CNCtxFactory`

Provider `URL: iiop://localhost:1050`

Stub Class: Click **Browse** and add the following jar file: `<public user documents>\Silk Performer 20.5\SampleApps\Java\Lib\sampleRmiOverIiop.jar`.

Sample Test Projects

The following sample projects are included with Silk Performer. To open a sample test project, open Silk Performer and create a new project. The **Workflow - Outline Project** dialog opens. Select the application type **Samples**.

.NET Sample Projects

.NET Remoting

This sample project implements a simple .NET Remoting client using the Silk Performer .NET Framework. The .NET Remoting test client, written in C#, comes with a complete sample .NET Remoting server.

Web Services

This sample shows you how to test SOAP Web Services with the Silk Performer .NET Framework. The sample project implements a simple Web Services client. The Web Services test client, written in C#, accesses the publicly available demo Web Service at: *http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx*

Java Sample Projects

JDBC

This sample project implements a simple JDBC client using the Silk Performer Java Framework. The JDBC test client connects to the Oracle demo user *scott* using Oracle's "thin" JDBC driver. You must configure connection settings in the `databaseUser.bdf` BDL script to run the script in your environment. The sample accesses the EMP Oracle demo table.

RMI/IIOP

This sample project implements a Java RMI client using the Silk Performer Java Framework. The test client uses IIOP as the transport protocol and connects to a RMI server provided as a sample application. For detailed instructions on setting up this sample project, see `<public user documents>\Silk Performer 20.5\SampleApps\RMILdap\Readme.html`.

The Java RMI server can be found at: `<public user documents>\Silk Performer 20.5\SampleApps\RMILdap`.

RMI

This sample project implements a Java RMI client using the Silk Performer Java Framework. The test client connects to a RMI server provided as a sample application. For detailed instructions on setting up this sample project, see `<public user documents>\Silk Performer 20.5\SampleApps\RMILdap\Readme.html`.

To access the Java RMI server:

If you have Silk Performer SOA Edition: Go to **Start** > **Programs** > **Silk** > **Silk Performer SOA Edition 20.5** > **Sample Applications** > **Java Samples** > **RMI Sample - SayHello** .

If you have Silk Performer Enterprise Edition: Go to **Start** > **Programs** > **Silk** > **Silk Performer 20.5** > **Sample Applications** > **Java Samples** > **RMI Sample - SayHello**.

Testing Java Components

The Java Framework encourages efficiency and tighter integration between QA and development by enabling developers and QA personnel to coordinate their development and testing efforts while working entirely within their specialized environments.

Testing Java Components Overview

With the Java Framework developers work exclusively in their Java programming environments while QA staff work exclusively in Silk Performer. There is no need for staff to learn new tools.

Java developers typically build applications and then hand them off to QA for testing. QA personnel are then tasked with testing Java applications end-to-end and component-by-component. Typically QA personnel are not given clients (test drivers) to test applications and they typically are not able to code such test clients themselves. This is where Java Explorer and the Java Framework are effective. Java Explorer offers a means of visually scripting test clients. In effect Java Explorer acts like a test client and can be used to interact with the application under test.

All Java components can be tested with Java Explorer, but the focus lies on the following three Java components: Enterprise JavaBeans (EJBs), Web Services, and Remote Method Invocation (RMI).

The Java Framework enables users to run stand-alone Java test code, to use other tools to invoke Java test code, or to execute test code from an exported standalone console.

Working With JDK Versions

Because multiple Java Developer Kit (JDK) versions are available, you need to test components against all versions. Both Silk Performer's Java Explorer and Java Framework support testing components of various vendors and of different JDK versions.

Java Framework Approach

The Java Framework approach to component testing is ideal for developers and advanced QA personnel who are not familiar with BDL (Benchmark Description Language), but are comfortable working with a Java development tool. With this approach, Java code is used by the QA department to invoke newly created methods from Silk Performer.

You can generate Java Framework BDL code using the Silk Performer JUnit import tool. The import tool produces BDL code that can invoke customer test code or customer JUnit testcode. It can also be used to directly invoke a client API.

Plug-In for Eclipse

Silk Performer offers a plug-in for Eclipse developers that automatically generates all required BDL code from within the Eclipse SDK. Developers simply write their code in Eclipse and implement certain methods for integrating with the Silk Performer Java Framework. The plug-in then creates all required BDL scripting that the QA department needs to invoke newly created methods from Silk Performer. The plug-in for Eclipse enables developers and QA personnel to better coordinate their efforts, consolidating test assets and enabling both testers and developers to work within the environments with which they are most comfortable.

Java Explorer Overview

Java Explorer is a GUI-driven tool that is well suited for QA personnel who are proficient with Silk Performer in facilitating analysis of Java components and thereby creating Silk Performer projects, test case specifications, and scripts from which tests can be run.

Developers who are proficient with Java may also find Java Explorer helpful for quickly generating basic test scripts that can subsequently be brought into a development tool for advanced modification.

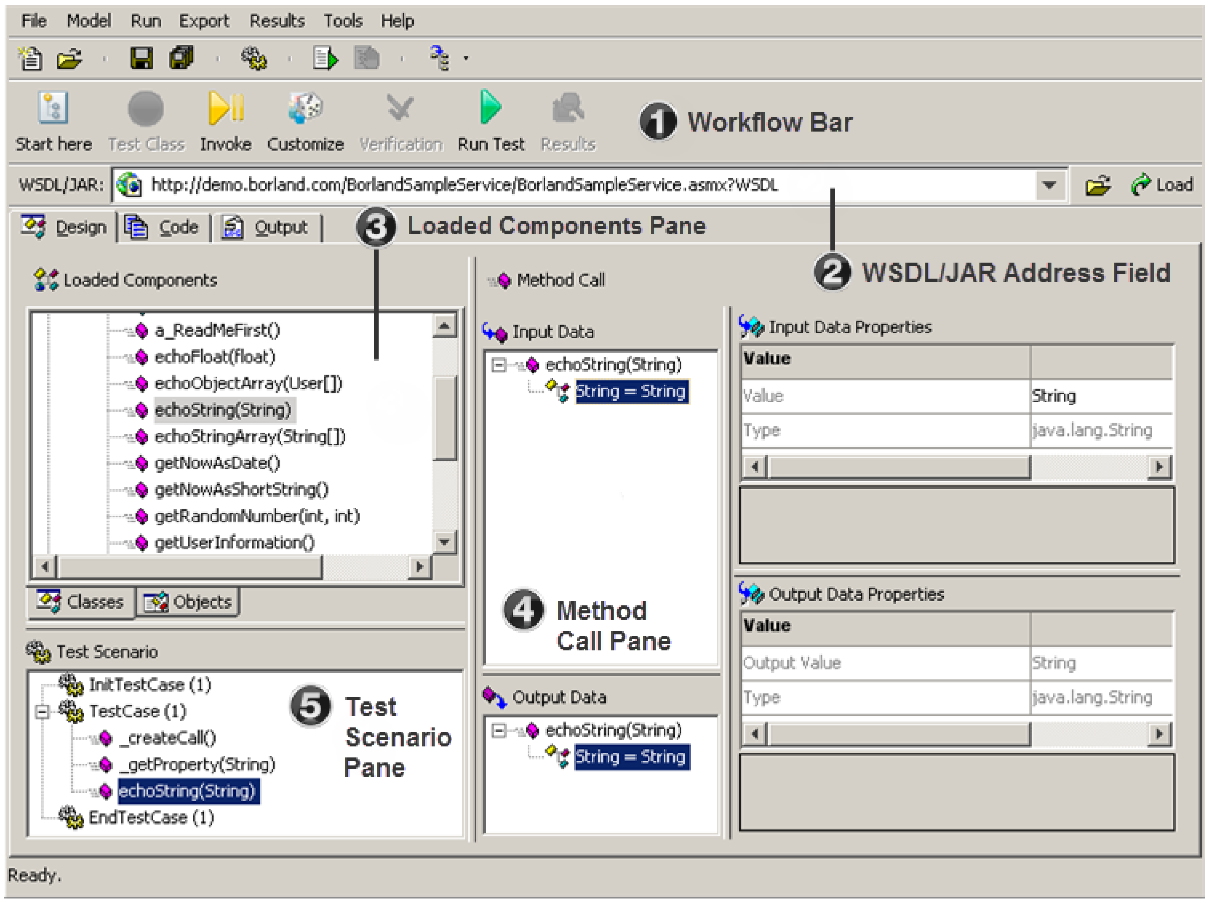
Java Explorer emulates Java clients. When working with Web services, Java Explorer achieves this through the use of proxies, which are conversion encoding/decoding modules between the network and client. Proxies communicate with servers by converting function calls into SOAP (XML) traffic, which is transferred through HTTP. Requests are decoded on the remote computer where the component resides. The XML request is then decoded into a real function call, and the function call is executed on the remote server. The results are encoded into XML and sent back to Java Explorer through SOAP, where they are decoded and presented to the user. What you see as the return result of the method call is exactly what the server has sent over the wire through XML.

Tour of the UI

Explains the view tabs that are offered by Java Explorer.

Overview

The main sections of the Java Explorer interface are highlighted below.



Workflow Bar

The Java Explorer workflow bar assists you in configuring and running your tests. The buttons on the workflow bar enable you to perform the following commands:

- **Start Here**: Opens the **Load File Wizard**, which guides you through selecting the type of testing that you want to perform and loading the required components.
- **Test Class**: Allows you to create a new test case for a class, or to add methods to an existing test case. This button is enabled when you select a class in the **Loaded Components** menu tree.
- **Invoke**: Invokes the selected method in the **Loaded Components** menu tree and adds it to the selected test case in the **Test Scenario** menu tree.
- **Customize**: Lets you store data as variables. This button is enabled when you select an input parameter in the **Input Data** menu tree, or an output parameter in the **Output Data** menu tree.

- **Verification**: Allows you to define verifications for variables. This button is enabled when you select an output parameter in the **Output Data** menu tree that has been stored as a variable.
- **Run Test**: Performs an animated run of your test cases.
- **Results**: Lets you explore the results of your last test run through user report, TrueLog, error report, or log file analysis.

WSDL/JAR Address Field

The **WSDL/JAR** address field allows for the loading of either WSDLs or Java classes and packages.

Loaded Components Pane

The **Loaded Components** pane includes three tabs:

- **Classes**: A list of loaded Web services, remotable classes or interfaces, and other Java classes. Allows you to instantiate objects, connect to remote objects, or call static methods.
- **Objects**: A list of instantiated objects, global variables, and random variables. Allows you to call methods on those kind of objects.

Method Call Pane

- The **Input Data** menu tree shows input parameters (both headers and bodies) for the selected method call.
- The **Input Data Properties** field shows the properties of the input parameters (value, type, pass, null, etc).
- The **Output Data** menu tree shows the output parameters (header and body) of the last/selected method call.
- The **Output Data Properties** field shows the properties of the output parameters (value, type, pass, null, etc).

Test Scenario Pane

The **Test Scenario** pane lists the current test cases along with all added method calls.

Design Tab

The **Design** tab is the primary mode used to access the majority of Java Explorer functionality. The **Design** tab offers the display of loaded components, invocation of methods, customization of input/output data, and the design of test scenarios.

Loaded Components Pane

The **Loaded Components** pane, on the **Design** tab, distinguishes between classes and objects and displays these components within different tree menus.

On the **Classes** tab you see all loaded classes with their members (methods, fields, and properties) that can be accessed without the need of an object context. Normally these are static members and constructors.

When you instantiate a class, the created object is shown on the **Objects** tab with all instance members (methods, fields, and properties).

The two exceptions are Web services and Remote Objects. As an object instance is not required for Web Services (which are objects that expose static methods), all methods offered by Web services are shown on the **Classes** tab. Remote Objects, for example EJB home objects, which are used to create an EJB object, are also listed in the **Classes** tab, although they are in fact objects.

Classes Tab

All loaded classes and Web services are listed in the Java Explorer **Loaded Components** pane on the **Classes** tab.

Only constructors, static methods, and members are displayed, as there is no object context required to call them.

Web services and remotable interfaces are treated as if they only have static methods.

The following folders are available in **Classes** view:

Remote Objects

Home interfaces of EJBs on remote servers. The home interfaces are looked up on a naming service through JNDI. Invoking a create method on EJB home objects instantiates EJB objects on servers.

Web Services

Stub classes, which are helper classes that encapsulate network communications. Axis classes are derived from `org.apache.axis.client.Stub` or `org.apache.axis2.client.Stub`, depending on the selected Web service plugin in **Tools** > **Project Settings** > **Web Service Plugin**. When you use the Metro client stub, classes are derived from `javax.xml.ws.Service`. Asynchronous method calls are not allowed and therefore are not shown.

Other Classes

Other Java classes, that do not fit in the above listed categories, and the application under test (AUT) are included in this folder.

Objects Tab

When an object is instantiated through a constructor or returned by a method and stored in a variable, the object is listed in the **Loaded Components** pane on the **Objects** tab. Instance members and methods that can then be invoked by the user are displayed.

The **Objects** tab shows objects stored in variables that have global scope and those that have a local scope of the currently selected test case.

Objects

Objects contain logic and data. For example, the data of an object might consist of a string. In this case, the logic of the object would consist of functions that can be used to manipulate the string.

Note: You can create new objects by invoking a method more than once using different input parameters.

Instance methods and members are displayed to allow for easy invokation.

Members have `get/set` accessors to enable the getting and setting of values.

Local test case objects are included in the `Local Test Case Objects` folder. A local object is only displayed while the test case that is associated with it is selected. Global test case objects are included in the `Global Test Case Objects` folder.

Note: To create a new test case, right-click a test case and choose **New Test Case**.

Variables

By default variables that store simple data types such as strings and integers are listed in the `Global Random Variables` folder, not as objects but as simple variables. Depending on the variable type - either simple value type or complex object - variables are either displayed as simple types, with no actions allowed, or as objects for which you can invoke all methods on the variable object, for example methods on `java.lang.String`.

String values are represented as complex objects that allow you to call all methods of the `java.lang.String` class.

Code Tab

Displays generated code files for the current test scenario. The following code types are available:

Test Driver Test Code

This tab displays the Java code generated for the test driver. This code can be compiled to a standalone console application. The main method for the application contains the calls to the test cases that are defined in your current test scenario. The test case methods contain all the calls that you have defined in your scenario for the respective test case.

When you export a standalone console application through the **Export** menu and then run the application, you see output information regarding successful/failed methods in the console window.

Silk Performer Java Framework Test Code

When you export a Silk Performer Java Framework project, the following files are generated:

- Silk Performer project file (.lpt)
- Silk Performer script file (.bdf)
- Java test script (.java)

This tab displays the Java test script code for Silk Performer Java Framework. Java framework code run by Silk Performer enables you to execute Java scripts through BDL. For each test case in your current test scenario, Java Explorer generates a separate BDL transaction in the test script.

BDL Script

This tab displays the BDL script code for Silk Performer Java Framework. The generated BDL executes the Java test script with the settings stored in the Silk Performer project file.

JUnit Code

This tab displays the JUnit test script code. this code can be executed with any JUnit TestRunner. For more information about JUnit testing, refer to the *Silk Performer Help*.

Client Proxy

This tab displays generated Web Service Client proxy code. When you load a WSDL file to access a Web service, a `Client proxy` class is created. Proxy classes are created on method and type information in WSDL files. When Web-service methods return complex types, additional classes are declared. These additional classes are used to represent complex types in the method calls that take complex parameters.

Output Tab

The **Output** tab displays log output during test runs. The tab shows details about each call of the most recent animated run. Statistics regarding execution time and executed methods are also shown.

Note: The **Output** tab can also be accessed by clicking **Results** on the Workflow Bar.

The following items are available:

Animation Log

Displays detailed information about test duration, method calls, and errors. It also lists each method, including input parameters and output values.

Java Explorer automatically saves this file to `<my documents>\Silk Performer 20.5\Projects\<projectname>\RecentTestRun\<projectname>.log`.

Error Log

The Error Log delivers detailed information about errors that occurred during the last test run.

Java Explorer automatically saves this file to `<my documents>\Silk Performer 20.5\Projects\<projectname>\RecentTestRun\<projectname>.err`.

User Report

An overview report in HTML format that displays the consolidated information from the animation log and the error log, in a report format that you can use for distribution or printing.

Java Explorer automatically saves this file to `<my documents>\Silk Performer 20.5\Projects\<projectname>\RecentTestRun\<projectname>.html`.

Example Animated Log Output

```
Test started: 08.10.2004 08:01:50
Test stopped: 08.10.2004 08:01:53
Method calls: 1
Success calls: 1
Error calls: 0
TestWebProxies.Service1.echoStringArray
Parameters : System.String[](string)
Return Value: System.String[](string)
```

Setting Up Java Explorer

This section describes how you can set up a project in Java Explorer and how you can define tests for the project. A sample Web Service host is used for demonstration purposes. For additional information on how to use Java Explorer to test Enterprise JavaBeans and RMI over IIOP, see *Enterprise JavaBeansand RMI over IIOP*.

Java Explorer Prerequisites

To work with Java Explorer you need to have a Java Development Kit 7, 8, downloadable from the *Java SE Downloads* page, or a IBM Java Development Kit installed on your system.

If you are using Microsoft Windows Vista or Microsoft Windows 7 as your operating system, you must have administrator priviledges to launch Java Explorer from the **Start** menu. If you do not have administrator priviledges, Java Explorer is not able to create files or folders or to run Silk Performer.

Some Web Services only work with specific SOAP stacks, depending on the server-side implementation of the Web Service. In this case, use trial and error to find the appropriate SOAP stacks, because the reported error messages are not clear.

Creating a New Project

To create a new Java Explorer project:

1. Start Java Explorer. The **Create or open a project** dialog box opens.
2. Click the **Create a New Project** option button.
3. Type a name for the project into the text box.

   Note: The project name cannot be longer than 19 characters and cannot contain certain special characters, for example `*`, `.`, or `!`.
4. Click **OK**. The new project is created and the Java Explorer Web GUI opens. A new folder, which contains all the files required by the project for operation, is created in the project directory. The default project directory is `<my documents>\Silk Performer 20.5\Projects`.
5. Navigate to **Tools** > **Project Settings** > **Web Service Plugin** .
6. Select the Web service plug-in (SOAP stack) that is required for your tests.

   Because of incompatibilities between different SOAP stacks, you cannot change the SOAP stack once it is used by Java Explorer. Web Service plug-ins that are used, are grayed out in the list box.

Test Case Definition

Use one of the following two approaches to define a test with Java Explorer:

- Use the **Load File Wizard** to define where the information for an object you want to test is located. The wizard displays the available classes, automatically instantiates the available objects, and calls methods on those objects. This approach requires more involvement, but provides more control.
- Enter the location of the Web services' WSDL file into the **WSDR/JAR** text box in Java Explorer and click **Load**. Information about the methods and objects is then displayed. This approach provides a quick and simple way to analyze remote service functionality.

Note: Objects are not instantiated and methods are not called.

For demonstration purposes this Help uses the **Load File Wizard** to instantiate the objects available on the sample Web service, which is available at *http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx?WSDL*.

Setting Up a Test Using the Load File Wizard

To set up a Java Explorer test using the **Load File Wizard**:

1. Click **Start Here** on the workflow bar. The **Load File Wizard** dialog box opens.
2. Click the option button that corresponds to the class type that you want to test.
   - **Web Services**
   - **Enterprise JavaBeans**
   - **RMI**
   - **Java Archive**
   - **Java Class**
3. To test a Web service, you must provide a Web Services Description Language (WSDL) document. Click the **Web Services** option button and specify a path or URL to the WSDL document using the browse **(...)** button.

   Note: To test Enterprise JavaBeans, click the corresponding option button and then specify the EJB-specific connection settings. To test other class types, click the appropriate option button and then specify the path to a Java archive or class file.
4. Click **Next**. The WSDL file now loads and finds out what is available on the Web service. A WSDL file is an XML description of the functionality that is offered by the Web Service.

   Note: The sample Web service offers a minimum of functionality-it simply echoes back some values.
5. Select the Web service class you wish to test from the **Class** list box. Normally there will be only one class, but it is possible that the loaded WSDL will define multiple Web services.
6. Use the **URL (Endpoint)** text box to change the endpoint of the service.

   The default endpoint is the one defined in the WSDL document.
7. *Optional:* To change the Web service proxy and authentication settings for the Web server, click browse **(...)** to open the **Connection Settings** dialog box.
   a) Type the endpoint of the Web service in the **URL** text box.
   b) If user credentials are required for authentication, type the information in the **Username** and **Password** text boxes.
   c) Click **OK**.
8. Click **Next**. The **Method Invocation** dialog box opens.
9. Check the check boxes beside the methods that you want to test.

   For example `echoFloat` and `echoString`. Methods are called using the default parameters.

   Note: The checked **Inherited** check box indicates that the member functions of the base classes are visible. You cannot change the state of the check box manually.
10. To adjust the sequence in which the methods are called, use the up and down arrow buttons. This way, you can move methods up and down the invocation list. The button between the up and down arrow buttons enables you to invert the selection of methods, meaning that all unmarked methods become marked, and vice versa.

    Note: Login and logout methods must be in first and last positions, respectively.

11. Click **Next**. The **Select a test case** dialog box opens.
12. Select the test case to which the testing methods should be added. You can select from the following:
    - **Init Test Case**: This is the first test case that is called in test runs. Select this test case if your method calls are related to initialization, logging in, or start-up.
    - **Existing Test Case**: You can add to an existing test case. Choose the test case from the list box to the right of this selection.
    - **New Test Case**: A new test case is created and the methods are added to the new test case. Using the text box to the right of this selection, give the new test case a unique name.
    - **End Test Case**: This is the last test case that is called in test runs. Select this test case if your method calls are related to clean-up, for example logging off.
13. Click **Finish**.

    Note: A message displays if you have selected a method that takes parameters, letting you know that default parameters will be used for the method, which may result in an exception being thrown. Methods that throw exceptions will be excluded from your test scenario. If you don't want to proceed, click **No**, then go back, deselect the method in question, and add it later manually with a more meaningful parameter value.

The Web service has been loaded in the class tree list and the selected methods have been called and added to the selected test case.

Test Case Characteristics

Following is an overview of the characteristics of test cases as they relate to Java Explorer:

- Each test scenario includes an `Init` test case and an `End` test case. Multiple `Main` test cases may also be defined. An `Init` test case is the first test case that is called during a test run. `Init` test cases often include method calls that are related to initialization, logging in, or start-up. An `End` test case is the last test case that is called during a test run. `End` test cases often include method calls that are related to clean-up, for example logging off.
- `Main` test cases are executed between `Init` and `End` test cases and may include any sort of method call.
- A local variable is only valid within a single test case and cannot be passed to another test case. Only global variables can be used to pass values between test cases.
- Each test case must have a unique name.
- Names of test cases are restricted. The names are mapped to Java methods and Silk Performer transactions, so they are subject to the restrictions of Java method and Silk Performer transaction names. Code exported to Silk Performer cannot use keywords that are defined by BDL or Java. You cannot use the names of the following variables because they are scripted by Java Explorer scripts when exporting to Silk Performer Java Framework Projects:

all	allownull	and	array	aux										
begin	benchmark	bin	boolean	by										
char	chr	clock$	commit	con	const	create	com1	...	com9	cursor				
database	dclevent	dclform	dclfunc	dclparam	dclrand	dclsql	dcltrans	dcluser	delete	dll	do	double	drop	dstring
else	elseif	end	eos	exit	explicit									
false	fetch	float	for	form	from	function								

group														
halt	handler													
if	in	init	inout	insert	into									
last	lenspec	long	loop	lpt1	...	lpt9								
mod														
next	not	nul	number											
of	optional	or	ord	order	out									
prn	proc	ptr												
rc	reset	result	return	rndbin	rndefxp	rndexpn	rndfile	rndind	rndpern	rndsno	rndstr	rndstream	rndunif	rndunii
rndunin	rollback	row	rows											
select	set	short	sizeof	sizespec	sql	stored	string							
then	throw	to	transaction	transactions	true									
union	unique	unsigned	update	use	user									
values	var	void												
wait	where	while	write	writeln										

- Test case properties can be edited by selecting a test case in the **Test Scenario** menu tree and defining field values in the **Output Data Properties** field:
  - `StopAtError` is the attribute that defines the number of errors that can occur during an animated run. If this number is reached, the animated run aborts. Setting this attribute to `0` specifies that errors will be ignored.
  - `CallCount` is the attribute that defines the number of times that a test case will be called during an animated run.
  - The `Name` attribute of the test case must be unique among all test cases.
- You can create, copy, and remove test cases by right-clicking a teste case in the **Test Scenario** menu tree and selecting the respective menu item.

Setting Up Tests

This section explains how to set up and customize your tests with Java Explorer. Topics include an introduction to GUI elements, customization of input parameters, definition of output value verifications, and generation of random variables. The section concludes with an explanation of how to use Java Explorer to test EJBs and RMI over IIOP.

Customizing Input Parameters

You should customize the input parameters before you execute a method. If you want to customize a method call after it is executed, for example if the method call was generated using the **Load File Wizard**, you need to execute an *Animated Run* after you have customized the method call to make the changes effective.

When a method has input headers or parameters, its values are displayed in the **Default Input Parameter Values** pane with their default values. Select the method you want to invoke in the **Loaded Components** pane. Parameters are displayed with their default values in the **Default Input Parameter Values** pane. To configure default values in the **System Settings**, click **Tools** > **System Settings** > **Default Values**.

To change a value, select a parameter in the **Default Input Parameter Values** pane. If a parameter is a simple data type, like string, float, or integer, you can enter the value directly into the text boxes in the pane. If the parameter is a complex type, like an object or array, or you want to use a random value or a value from a stored variable, click **(...)** to open the **Input Value Wizard**.

To use the **Input Value Wizard** to customize an input parameter:

1. Select the `InputValue` row in the **Input Data Properties** pane.
2. Click **(...)**. The **Input Value Wizard** opens.
3. Define the input value for the parameter of the method.

   You can pass a null value, a value from a variable, or a constant value.

   - Click**Is Null** to pass a null value. This value is only valid for strings, objects, and arrays.
   - If you have already stored a variable that holds values of the parameter's type, click **Use Variable** and select the variable from the list box.

     If the parameter is an array, type the index into the **Index Array** text box.
   - If the parameter is not an object or array, you can specify a constant value such as a string, integer, or double value. Select the **Constant value** option button.
   - If you do not want to use a constant value and would rather use a random value for types such as string, integer, or float, you can create a new random value by clicking **New random variable**. If a random value of the parameter type has already been created, click **Random variable** and select the variable from the list box.
4. Click **OK**.

Once input parameters have been defined, you can invoke a method to have that method automatically added it to the selected test case in the **Test Scenario** pane. From there, corresponding output values can either be stored as variables or verifications can be set up for them.

Storing Output Values in Variables

Using the **Output Data** and **Output Data Properties** panes, you can store output values as variables for later use or you can define verifications for the values.

1. Click **Invoke** on the Workflow bar or by right-click a method in the **Input Data** pane and choose **Invoke/Add to Test Case** to invoke the method. The **Output Value Wizard** opens.

   Note: You can also open the **Output Value Wizard** by clicking **Customize** in the Workflow bar, or by right-clicking an output parameter in the **Output Data** pane.
2. In the **Output Value Wizard**, type a name for the output value variable in the **Variable name** text box.
3. Click the option button that corresponds to the scope of the variable.

   The default is **Local**, which means that the variable can only be used within the test case within which it has been created. **Global** scope means that the variable can be used within any test case. Global scope enables you to pass information between test cases.
4. Click **OK**.

Manually Adding Global Variables

Global variables can be shared between test cases. They vary from *local variables*, which are only available to specific test cases.

Note: You can also create global variables by clicking the **Global scope** option button in the **Output Value Wizard**.

Global variables can be used as input parameters for function calls. Global variables are configured with an initial value, which makes them useful when you need to have the same value used for multiple function calls. Each time a test run begins, global variables are initialized with the defined initial value.

To create a global variable:

1. Click the **Objects** tab.
2. Right-click an object in the **Objects** tree menu and choose **New Global Variable**. The **Define a Global Variable** dialog box displays.
3. Enter a name for the variable in the **Variable Name** text box.
4. Select a **Variable type**.

   The following variable types are available:

   - boolean
   - byte
   - short
   - int
   - long
   - char
   - float
   - double
   - java.lang.String
5. Each variable has an initial value and a current value. When tests are run, all global variables are initialized to their initial values. Specify an intial value for the variable in the **Initialize Value** text box.
6. Click **OK**.

Capturing an HTTP Response

By default, Java Explorer does not allow you to view the HTTP response when running a test against a Web service. As an alternative, you can use Silk Performer to record and capture the server response.

1. Launch Silk Performer.

2. In the Silk Performer menu, click **Settings** > **System** .
3. Click the **Recorder** icon. The **Recording Profiles** page opens.
4. Click **Add** to add a new recording profile for Java.

   The **Recording Profile** dialog box opens.
5. In the **Profile name** text box, type a unique name for the Java recording profile.

   The profile name is required to identify the application in the Silk Performer Recorder window.
6. In the **Application path** text box, type the name of the Java application's executable and the directory in which it resides.

   To locate the executable, click **Browse**.

   Note: There may be multiple Java executables. Therefore, it is important that the Recording Profile should match the Java executable used in the Java Explorer runtime settings.
7. In the **Working directory** text box, type the working directory of the Java application.

   The working directory is the folder that contains the application or related files. Some applications need files located in other directories, so you might have to specify the folder in which these files reside.
8. In the **Application type** list box, make sure that **Custom Application** is selected.
9. In the **Protocol selection** area, select the **Web** check box, which identifies that the Silk Performer Recorder hooks into a Web application, intercepts all function calls, and displays the results.
10. Click **OK** to add the new Java profile to the list.
11. Ensure that you record the script in Silk Performer with the newly created recording profile before invoking and running the test in Java Explorer.
12. After you save the recording, you can view the HTTP response in TrueLog Explorer by selecting the **In Body** tab of the recorded TrueLog file (`.xlg`).

Defining Output Value Verifications

1. Select **Verification** in the **Output Data Properties** pane to display the **(...)** button.
2. Click **(...)** to open the **Verification Wizard**.
   a) Select **Range** if you are verifying integers or doubles. Then specify a range for the value using the **From** and **To** text boxes.
3. Select one of the following:
   - Click **No Verification** if you do not want the value to be verified.
   - Click **Verification with NULL or variable** to verify one of the following:
     - Check the **Should be null** check box, to verify that the value is NULL.
     - The value is not NULL. This option verifies whether or not a value is present.
     - Verify that the value exactly matches the value of an existing variable, a public member of a variable, or an element of an array.If the variable is an array you can also define the **Array index** for the element that should be used for the verification.
   - Click **Constant Value** to verify that the value exactly matches a constant string.
   - Click **Regular expression** to verify against a string value.
   - Click **Range** if you are verifying that a numeric value is within a specified range. Then specify a range for the value using the **From** and **To** text boxes.
4. Select the **Severity** of error that should be thrown if the verification fails. The following severity options are available:
   - `Success, no error`

     No error is thrown
   - `Informational`

An informational message is logged

- `Warning`

  A warning is thrown
- `Error, simulation continues`

  An error is thrown, but the simulation continues
- `Error, the active test case is aborted`

  An error is thrown, the simulation continues, however the active test case is aborted. This is a transaction exit severity in Silk Performer
- `Error, the simulation is aborted`

  An error is thrown and the simulation is aborted
- `Custom`

  A section is inserted into the .NET code where special error handling code can be inserted

5. Click **OK**.

Verification with NULL or Variable

The **Verification with NULL or variable** selection inludes a list box that contains the entries < is Null > and < is not Null > and a list of variables that match the return value of the method call.

The list box additionally contains arrays for which the element type matches the return value of the method call. For example, the array `myStringArray` has a return value of type `java.lang.String`. You can select variables that represent a string array from the list box. In such cases, an array index list box is enabled.

The list box also contains public fields of complex objects of matching type. For example, the test class `PublicMember` has a public field called `mStr` of type string, which can be selected. The notation of public fields is *<variable name>.<member name>*, which for the example is *PublicMember.mStr*.

Adding Method Calls

1. Within the **Loaded Components** or **Method Call** pane, select the method to be added.
2. Invoke the method by clicking **Invoke**. The invoked method is added automatically to the currently selected test case. If the method call returns a value, the **Output Value Wizard** opens automatically, and prompts you to store the value.

Updating Method Calls

1. From the **Test Scenario** pane, select the method that is to be changed.
2. Make modifications to the input/output parameters of the method in the **Input Data** or **Output Data** panes. Updates are automatically accepted when you exit the **Input Data** or **Output Data** panes.
3. Perform an *Animated Run* to apply the changes.

Complex Input/Output Data

Java Explorer supports complex data types such as arrays and objects as both input data and output data. The **Output Value Wizard** assists you in using such data as variables.

Verification of Variables

Java Explorer fully supports the verification of complex data types. When the return value of a method, including array elements and public fields of complex type, is complex, you can select an existing variable, an array element of an existing variable, or a public field of an existing varible to verify against. Java Explorer also supports the verification of variables with primitive data types.

Storing Array or Object Output Data as Variable

When a method returns an array or an object as output data, you can store the value in a variable of either local or global scope.

1. Select the complex output parameter in the **Output Data** pane.
2. Click **(...)**. The **Output Value Wizard** opens.
3. Specify the name and scope of the variable.

Taking Array Object Data From a Variable

1. Select a complex input parameter in the **Input Data** pane.
2. Click **(...)**. The **Input Value Wizard** opens.
3. Define the input value.

Defining Each Member of a Variable Individually

1. Select individual members of an object in the **Input Data** pane.
2. Define input values in the **InputValue** text box.

   For example, you can specify a random value from a value list as one input value and specify a constant value as another input value.

Defining Array Elements

1. The default length for arrays is `1`. To change the number of elements in an array, select the array node in the **Input Data** pane.
2. In the **Length** text box, enter the number of elements in the array. Java Explorer automatically adds or removes the required elements.
3. To specify the value of an element, select the element in the array and specify the value in the **Value** text box.
4. Array members or an entire array can be assigned input parameters from a variable. To specify input parameters from a variable, click **(...)**. The **Input Value Wizard** opens.
5. Click **Use variable** on the **Input Value Wizard** and select a preconfigured variable from the list box.

   All variables of the current parameter type are listed.
6. Click **OK**.

Using Complex Input Data

The following example task describes how you can use complex input data:

1. Call the method `echoStringArray` of the sample Web service.

The sample Web service is available from *http://demo.borland.com/BorlandSampleService/BorlandSampleService.asmx?WSDL*. This service requires an object of the generated class `ArrayOfString` as a parameter.

2. Right click **init()** and choose **Invoke** in the **Design** pane to instantiate the `ArrayOfString` object. The **Output Value Wizard** opens.
3. Check the **Store as variable** check box.
4. Type `myArrayOfString` into the **Variable name** text box.
5. Click **OK**. The **Output Value Wizard** closes.
6. Right click **setString(String[])** and choose **Invoke** in the **Design** pane to call the `setString` method and define the array elements.
7. Use the **Input Value Wizard** to pass `myArrayOfString` to the `echoStringArray` method.

Random Value Types

When you define a random variable for an input value you can only choose one of the random types that are applicable for that input value type.

Random variables return one of the following value types:

- `String`
- `Number`
- `Float`

String Random Values

You can define random variables that return strings with the **Random Variable Wizard**.

Individual Strings

An *individual string* declares a random variable of type `RndInd`. With each access, such a variable contains a random value with type `string`. You must declare all possible values for the random variable, along with their probabilities. For each string value you must define a weighted number that indicates the probability that the string value will be selected. Weighted numbers are integer numbers.

Strings from a Pattern

A *string from a pattern* declares a random variable of type `RndStr`. Such a variable contains a random string value with each access. The characters of the string and the length of the string are randomly generated following a uniform distribution.

The pattern string defines the characters that you must use to construct random strings. The length of calculated strings is selected randomly with a uniform distribution based on defined minimum and maximum lengths, including boundaries.

Number Random Values

You can define random variables that return integers with the **Random Variable Wizard**. The values can either be calculated based on a weighted number list or by specifying a number range.

Integer Numbers

Declares a random variable of type `RndUniN`. With each access, such a variable contains a value that is generated following a uniform distribution. The parameters of the `RndUniN` function specify the minimum value and the maximum value of the uniform distribution. Within these boundaries all integer values have the same probability, including the boundaries.

Weighted Integer Numbers

Declares a random variable of type `RndInd`. With each access, such a variable contains a random value whose type is either string or integer, depending on the syntax of the declaration. You must declare all possible values for the random variable, along with their probabilities. You cannot mix integer and string values. For each integer value you must define a weighted number that indicates the probability that the integer value will be selected.

Float Random Values

You can define random variables that return floats with the **Random Variable Wizard**. There is only one possible definition for generating float values currently.

Floating Point Numbers

Declares a random variable of type `RndUniF`. With each access, such a variable contains a value that is generated following a uniform distribution. The parameters of the `RndUniF` function specify the minimum value and the maximum value of the uniform distribution.

Configuring Random Variables

With the **Random Variable Wizard**, you can configure random variables for entire input parameters or individual members of complex input parameters. You can also reuse existing random variables by using the **Input Value Wizard**.

To configure a random variable:

1. On the **Input Value Wizard**, click the **New random variable** option button.

   Alternative: If you have already configured a random variable that you would like to apply, click the **Use random variable** option button.
2. *Alternative:* If you have already configured a random variable that you would like to apply, perform the following:
   a) Click the **Use random variable** option button.
   b) Select the variable that you wish to apply from the list box.
   c) Click **OK**.
3. Click **OK**. The **Random Variable Wizard** opens.
4. From the **Random type** list box, select the type of random variable that you wish to insert to your script.

   A short type description and a variable declaration preview for each type is displayed.

   Note: Random variables always have global scope, and so are available to all test cases in your project.
5. Click **Next**.
6. In the **Parameter** pane, edit the existing valyues or insert new values.

   Weight numbers define the fraquency at which values are displayed. The weight numbers are more flexible than using percentages to define frequency.
7. Click **Finish**.

Enterprise JavaBeans and RMI over IIOP

Enterprise JavaBeans (EJB) is a specification for developing scalable, server-side, multi-user enterprise applications. When you test components like EJBs or other remote components, you cannot use HTTP traffic when the application expects RMI or RMI over IIOP traffic. To communicate with the application, Silk Performer must invoke a Java client that generates the appropriate protocols that can be used to approach the interface of the application.

Java Remote Method Invocation (RMI) technology is developed jointly by Sun Microsystems and IBM. RMI delivers CORBA-distributed computing functionality to the Java 2 platform, when the technology is run over Internet Inter-Orb Protocol (RMI-IIOP).

Working with Enterprise JavaBeans

Testing an EJB with Java Explorer involves the following three main steps:

- Connecting to the Java Name Service (JNDI)
- Selecting an EJB Home object that is registered at the Naming Service and can be used as the factory object
- Creating an EJB on the application server

To test an EJB:

1. In the Workflow bar, click **Start Here**. The **Load File Wizard** opens.
2. *Alternative:* In the **Classes** pane, right-click **Remote Objects** and choose **New EJB** to start the **EJB Wizard** directly.

   In this approach, skip the next two steps.
3. Click the **Enterprise JavaBeans / RMI over IIOP** option button.
4. Click **Next**. The **Connect to Naming Server** dialog box opens.
5. If available, select the application server vendor from the **Server** list box.

   Note: The entries in the list work as a template that pre-configures other settings, like initial context factory, protocol, and port, with default values. For additional information regarding the configuration of specific application server types, see *Vendor-Specific JNDI Settings*.
6. In the **Factory** text box, specify the context factory of the naming service.
7. In the **Host/Port** text boxes, type the URL where the naming service provider is to be hosted.

   The naming server provider URL is comprised by protocol, host, and port. You can also type the URL directly into the **Provider URL** text box.
8. If required by the provider, type the credentials in the **User** and **Password** text boxes.
9. Click **Edit Classpath ...**. The **Classpath Configuration** dialog box opens.
10. Choose **Static** and click **Add entries to classpath** to specify the archives that are necessary to connect to the naming service of the application server.
11. Choose Dynamic and click **Add entries to classpath** to specify the archives that contain the client stubs of the application.

    For vendor-specific details, refer to the application server documentation. For additional information on classpaths, see *Classpath Settings*.
12. Click **OK**.
13. Back on the **EJB Wizard**, click **Next** to connect to the naming service. The **Look up and narrow a Home Interface or Remote Object** dialog box opens.
14. In the naming service sub-folders, select the EJB Home interface, represented by a "hand holding a coffee bean" icon, or the remote object, represented by a satellite dish icon, for which Java Explorer should obtain a reference.

    The corresponding client stubs must have been added to the classpath of the project.
15. Click **Finish**.

    You can now use the **Load File Wizard** to browse for the methods of the remote object and to specify which test case the methods are to be called from.

    Note: EJBs which implement the `EJBLocalObject` interface instead of the `EJBHome` interface are not displayed correctly in the JNDI tree.
16. *Alternative:* If you invoke the **EJB Wizard** directly by right-clicking **Remote Objects** in the **Classes** pane, you must perform the following two steps:

a) Select the `create()` method of the EJB Home object to instantiate an EJB object. Store the EJB object to a variable.
b) Invoke one or more methods on the EJB object.

Vendor-Specific JNDI Settings

The application server vendor-specific default values are inserted automatically when you select the aproperiate server name in the **Connect to Naming Service** dialog box. The following vendor-specific default values exist:

- initial context factory
- protocol
- port

Note: The naming service provider URL is comprised by protocol, host, and port.

Connecting to BEA WebLogic

To connect to a BEA WebLogic server:

1. In the **Connect to Naming Service** dialog box, choose `WebLogic Application Server` from the **Server** list box.
2. From the **Factory** list box, choose `weblogic.jndi.WLInitialContextFactory`
3. In the **Provider Url** text box, type `t3://red:7001`.

   The port may vary depending on the configuration of the server.
4. Click **Edit Classpath ...** and specify the following server and client classpaths:
   a) In the **Server Classpath** text box, type the path to `weblogic.jar`.

      This file is provided by BEA and contains the WebLogic J2EE implementation and the implementation used for connecting to the naming mechanism of BEA WebLogic. The file has a size of about 40 MB and you can normally find it in a lib subdirectory of the BEA WebLogic home directory.
   b) In the **Client Classpath** text box, type all JAR files and classpaths that are required to connect to the EJB.

      You usually put the client-side stubs and additional classpath entries into the client classpath. WebLogic servers automatically generate the stubs when EJBs are deployed.
5. Verify your settings and click **Next**. The names of all registered EJBs and RMI objects that are hooked into your WebLogic server are browsed.
6. Choose the home interface that allows for the creation of instances of EJBs.

   It is typical for EJBs to have home interfaces that are used to create EJB instances. For example, choose **ejb20-statelessSession-Traderhome** to use the TraderHome interface.
7. Click **Finish**.
8. In the **Load File Wizard**, click **Next**. The home interface you have selected is browsed for the published methods it includes. A method named create is likely to be among these methods. Call this method to get a valid reference to an EJB object.
9. Click **Next** and select the test case where Java Explorer should write code for the steps that are performed by the wizard.

   For example, you can put the steps that are referred to as *bootstrapping* into the init test case or an existing test case.
10. Click **Next**.
11. Check the **Store as variable** check box and type a name for the EJB reference obtained by the **Java Explorer Wizard** into the **Variable name** text box.

12. Click **OK**. You can now call any of the EJBs business methods.

Connecting to IBM WebSphere

To connect to an IBM WebSphere server:

1. In the **Connect to Naming Service** dialog box, choose `WebSphere Application Server` from the **Server** list box.
2. From the **Factory** list box, choose `com.ibm.websphere.naming.WsnInitialContextFactory`.
3. In the **Provider Url** text box, type `iiop://lab44:900`.

   The port may vary depending on the configuration of the server.
4. Click **Edit Classpath ...** and specify the following server and client classpaths:
   a) In the **Server Classpath** text box, type the path to `csicpi.jar`, which is the IBM implementation of J2EE.

      This file is provided by IBM and contains the IBM J2EE implementation.

   b) Additionally type the paths to the following required packages into the **Server Classpath** text box:

      WebSphere 4.x
      - `ejbcontainer.jar`
      - `iwsorb.jar`
      - `j2ee.jar`
      - `jts.jar`
      - `ns.jar`
      - `ras.jar`
      - `ujc.jar`
      - `utils.jar`
      - `websphere.jar`

      WebSphere 5.x
      - `ffdc.jar`
      - `idl.jar`
      - `iwsorb.jar`
      - `j2ee.jar`
      - `naming.jar`
      - `namingclient.jar`
      - `ras.jar`
      - `wsexception.jar`
      - `ecutils.jar`
      - `tx.jar`
      - `utils.jar`
      - `ejbportable.jar`
      - The properties directory, which is usually located in the WebSphere home directory.

      These files are shipped with each WebSphere installation and are normally found in the lib directory of WebSphere installations.

   c) In the **Client Classpath** text box, type all JAR files and classpaths that are required to connect to the EJB.

      You usually put the client-side stubs and additional classpath entries into the client classpath. WebSphere servers automatically generate the stubs when EJBs are deployed. You can normally find these stubs in the installedApps directory, which is located within the WebSphere home directory.
5. Verify your settings and click **Next**. The names of all registered EJBs and RMI objects that are hooked into your WebSphere server are browsed.

6. Choose the home interface that allows for the creation of instances of EJBs.

   It is typical for EJBs to have home interfaces that are used to create EJB instances. For example, choose **cart** to use the CartHome interface from the petstore sample.
7. Click **Finish**.
8. In the **Load File Wizard**, click **Next**. The home interface you have selected is browsed for the published methods it includes. A method named create is likely to be among these methods. Call this method to get a valid reference to an EJB object.
9. Click **Next** and select the test case where Java Explorer should write code for the steps that are performed by the wizard.

   For example, you can put the steps that are referred to as *bootstrapping* into the init test case or an existing test case.
10. Click **Next**.
11. Check the **Store as variable** check box and type a name for the EJB reference obtained by the **Java Explorer Wizard** into the **Variable name** text box.
12. Click **OK**. You can now call any of the EJBs business methods.

Connecting to Sun

To connect to a Sun server:

1. In the **Connect to Naming Service** dialog box, choose `SUN J2EE Server` from the **Server** list box.
2. From the **Factory** list box, choose `com.sun.jndi.cosnaming.CNCtxFactory`
3. In the **Provider Url** text box, type `iiop://red:1050`.

   The port may vary depending on the configuration of the server.
4. Click **Edit Classpath ...** and specify the following server and client classpaths:
   a) In the **Server Classpath** text box, type the path to `j2ee.jar`.

      This file is provided by Sun and contains the J2EE implementation and the implementation used for connecting to the naming mechanism of Sun. You can normally find the file in a lib subdirectory of the J2EE home directory.
   b) In the **Client Classpath** text box, type all JAR files and classpaths that are required to connect to the EJB.

      You usually put the client-side stubs and additional classpath entries into the client classpath. Sun J2EE servers automatically generate the stubs when EJBs are deployed.
5. Verify your settings and click **Next**. The names of all registered EJBs and RMI objects that are hooked into your Sun server are browsed.
6. Choose the home interface that allows for the creation of instances of EJBs.

   It is typical for EJBs to have home interfaces that are used to create EJB instances.
7. Click **Finish**.
8. In the **Load File Wizard**, click **Next**. The home interface you have selected is browsed for the published methods it includes. A method named create is likely to be among these methods. Call this method to get a valid reference to an EJB object.
9. Click **Next** and select the test case where Java Explorer should write code for the steps that are performed by the wizard.

   For example, you can put the steps that are referred to as *bootstrapping* into the init test case or an existing test case.
10. Click **Next**.
11. Check the **Store as variable** check box and type a name for the EJB reference obtained by the **Java Explorer Wizard** into the **Variable name** text box.
12. Click **OK**. You can now call any of the EJBs business methods.

Connecting to JBoss

To connect to a JBoss server:

1. In the **Connect to Naming Service** dialog box, choose `JBoss Application Server` from the **Server** list box.
2. From the **Factory** list box, choose `org.jnp.interfaces.NamingContextFactory`
3. In the **Provider Url** text box, type `jnp://red:1099`.

   The port may vary depending on the configuration of the server.
4. Click **Edit Classpath ...** and specify the server and client classpaths.

   For additional information regarding the testing of JBoss application servers, contact *Customer Care*.
5. Verify your settings and click **Next**. The names of all registered EJBs and RMI objects that are hooked into your JBoss server are browsed.

Working with RMI Over IIOP

To test RMI over IIOP:

1. In the Workflow bar, click **Start Here**. The **Load File Wizard** opens.
2. Click the **Enterprise JavaBeans / RMI over IIOP** option button.
3. Click **Next**. The **Connect to Naming Service** dialog box opens.
4. If available, select the application server vendor from the **Server** list box.

   Note: The entries in the list work as a template that pre-configures other settings, like initial context factory, protocol, and port, with default values. For additional information regarding the configuration of specific application server types, see *Vendor-Specific JNDI Settings*.
5. In the **Factory** text box, specify the context factory of the naming service.
6. In the **Host/Port** text boxes, type the URL where the naming service provider is to be hosted.

   The naming server provider URL is comprised by protocol, host, and port. You can also type the URL directly into the **Provider URL** text box.
7. If required by the provider, type the credentials in the **User** and **Password** text boxes.
8. Click **Edit Classpath ...**. The **Classpath Configuration** dialog box opens.
9. Choose **Static** and click **Add entries to classpath** to specify the archives that are necessary to connect to the naming service of the application server.
10. Choose Dynamic and click **Add entries to classpath** to specify the archives that contain the client stubs of the application.

    For vendor-specific details, refer to the application server documentation. For additional information on classpaths, see *Classpath Settings*.
11. Click **OK**.
12. Back on the **EJB Wizard**, click **Next** to connect to the naming service. The **Look up and narrow a Home Interface or Remote Object** dialog box opens.
13. In the naming service sub-folders, select the EJB Home interface, represented by a "hand holding a coffee bean" icon, or the remote object, represented by a satellite dish icon, for which Java Explorer should obtain a reference.

    The corresponding client stubs must have been added to the classpath of the project.
14. Click **Finish**.

    You can now use the **Load File Wizard** to browse for the methods of the remote object and to specify which test case the methods are to be called from.

Note: EJBs which implement the `EJBLocalObject` interface instead of the `EJBHome` interface are not displayed correctly in the JNDI tree.

Testing the RMI Sample

To test the Silk Performer RMI sample:

1. Launch the *Product Manager* RMI sample.
2. Navigate to the `<public user documents>\Silk Performer 20.5\SampleApps\RMILdap` directory.
3. Invoke the four batch files `step1` through `step4`.
4. In the **Classes** pane, right-click **Remote Objects** and choose **New EJB**. The **EJB Wizard** opens. The RMI objects are registered at a JNDI naming service.
5. Type `com.sun.jndi.cosnaming.CNCtxFactory` into the **Factory** text box.
6. Type `iiop://localhost:1900` into the **Provider Url** text box.
7. Click **Edit Classpath ...** and select `<public user documents>\Silk Performer 20.5\SampleApps\RMILdap\RMILdap\ProductManager-server\ProductManager-server.jar`.
8. Back on the **EJB Wizard**, click **Next** to connect to the naming service. The **Look up and narrow a Home Interface or Remote Object** dialog box opens.
9. Select the *ProductManagerServicesLdap* remote object. The `RmiExtension` helper class performs all necessary bootstrapping.
10. Click the `RmiExtension` helper class in the **Classes** pane of the **Loaded Components** page.
11. Click the `createObject` method of the class in the **Input Data** pane.

    The method includes a class to provide the parameters for the RMI connection settings.
12. View the result of the `createObject` method in the **Objects** pane. The result is a stub class instance, which is used to communicate with the remote object on the server.
13. Open the `getProductsByName` method in the **Loaded Components** page.

    The method requires a string as parameter. You can use `*` as a wildcard to get all products.

Java RMI

Java Explorer supports the following protocols for Java RMI:

- RMI over IIOP
- Java Remote Method Protocol (JRMP)

RMI Testing with JRMP

The main difference between testing RMI with JRMP and testing RMI over IIOP is that with JRMP you need to connect to an RMI registry.

To test RMI with JRMP:

1. In the Workflow bar, click **Start Here**. The **Load File Wizard** opens.
2. *Alternative:* In the **Classes** pane, right-click **Remote Objects** and choose **New RMI** to start the **RMI Wizard** directly.

   In this approach, skip the next two steps.
3. Click the **RMI** option button.
4. Click **Next**. The **Rmi Wizard** dialog box opens.

5. In the **Host/Port** text boxes, type the name and port of the host.

   For example, type `localhost` and `1099`.
6. Click **Add archive(s) to classpath** and specify the location of the RMI interfaces for the client.

   For example, `<public user documents>\Silk Performer 20.5\SampleApps\Java\Lib\sampleRmi.jar`.
7. Back on the **EJB Wizard**, click **Next** to view all registered remote objects in the RMI registry. The **Look up and narrow a Remote Object** dialog box displays all registered remote objects in the RMI registry.
8. Choose **RemoteServer** and click **Next**. The methods of the remote object are browsed.
9. Select the first two methods by checking the appropriate check boxes.

   - `echoFloat(float)`
   - `echoString(string)`

   The selected methods are implemented by the sample RMI server. The other methods are inherited from base classes and don't need to be tested.
10. Click **Next** and select the test case from which the methods should be called.

Testing Java Archives

Java archives normally contain many classes that need to be tested.

To test a Java archive:

1. In the Workflow bar, click **Start Here**. The **Load File Wizard** opens.
2. Click the **Java Archive** option button.
3. Click **Browse ...** to browse for the JAR file that contains the classes to be tested.

   For example, load `sampleBasis.jar`. This archive is selected by default.
4. Click **Next**. The **Testing Java Classes** dialog box opens.
5. In the **Class** text box, type the name of the class to be tested.
6. Choose the constructor of the class from the **Constructor** list box.
7. Click **Next**. The instance of the class is created and the created object is browsed for the methods it includes.
8. In the **Method Invocation** dialog box, select the `toString` method.
9. Click **Next**.
10. Choose a test case where Java Explorer should add the created scenario. The method `toString` returns a string. Therefore, Java Explorer asks you whether it should store the returned value or discard it.

Test Class Wizard

You can create a test for a class or an instantiated object with the **Test Class Wizard**. To use the wizard, select a class in the **Classes** pane or an object in the **Object** pane and call the wizard, either through the context menu or the Workflow bar. You can also select methods that are to be called with the wizard.

Secure Web Services

You can configure Java Explorer for the testing of secure Web services.

To test secure Web services, the following steps are required:

- Create a truststore for your JDK.
- Load a secure WSDL into Java Explorer.
- Invoke Web service calls over a secure connection.

Java Explorer supports authentication of Web services. Use the **Connection Settings** dialog box to specify your credentials.

Creating a Truststore for Your JDK

When you work with secure Web services, the Web service clients must trust the Web applications they interact with to be secure. The Web service client is typically a Web browser, but in this case the client is Java Explorer. To enable testing of a Web application, Java Explorer must accept the server certificate of the application or trust the certifying authority who issues the server certificate.

Your JDK installation includes a command-line tool called *keytool*. You can manage server certificates and certifying authorities with this tool. The easiest way of downloading server certificates is to export them from your Web server.

To create a truststore for your JDK:

1. Export a `CER` certificate file from your Web browser to your local system.

   For information on exporting a certificate, refer to the Help of your browser.
2. To add the saved certificate to your truststore, which is called `mykeystore`, enter the following into the command-line tool of your system:

   ```
   keytool
   -import
   -alias <alias name>
   -file c:\<pathname>\<certificate file name>
   -keystore mykeystore
   ```

   The command-line tool creates a file called `mykeystore` in your current working directory.

Configuring Java Explorer to Use a Truststore

Set the system property `-Djavax.net.ss.trustStore=trustedcerts` to configure a Java process to use a truststore. When you work with Java Explorer, you can specify the system properties in property files that reside at `<Java Explorer Home>/Startup directory`.

To configure Java Explorer to use a truststore:

1. Open `./startup/gui.properties` in a text editor.
2. Add the following line:

   ```
   system.property.javax.net.ssl.trustStore=C:\\Temp\\mykeystore
   ```
3. Open `./startup/runtime.properties` in a text editor.
4. Add the following line:

   ```
   system.property.javax.net.ssl.trustStore=C:\\Temp\\mykeystore
   ```
5. Restart Java Explorer.

Loading a Secure WSDL

To load a secure WSDL document into Java Explorer:

1. Copy the URL of the WSDL file into the address bar of Java Explorer.
2. Press **Enter**.

Note: Ensure that you have configured Java Explorer to use a truststore that accepts the server certificate of the URL.

3. If loading the WSDL document fails, try the following:
   a) Open the WSDL document in a Web browser.
   b) Save the WSDL document to your local machine.
   c) Open the document with Java Explorer.

Working with Secure Web Service Calls

To execute Web service calls over a secure connection, ensure that you have created a truststore for your JDK and that you have configured the Java Explorer Runtime process to use a truststore.

Note: Use Apache Axis 1.4 or later to test secure Web services.

To configure Java Explorer to use Apache Axis:

1. Click **Tools** > **Project Settings**.
2. Click the **Web Service Plugin** tab.
3. Select `Axis 1.4` or `Axis2 1.5` from the **Assigned Web Service Plugin** list box.

   Note: The read-only **Classpath** pane displays the classpath configuration that belongs to the selected Axis version.

   The selected Axis version is stored persistently in the project file of Java Explorer.
4. Click **OK**.
5. Click **Yes** to confirm that you want to change the Web service library settings.

Specifying Web Service Connection Settings

To specify Web service connection settings:

1. In the **Loaded Components** pane, click the **Classes** tab.
2. Choose the loaded Web service class node to display the **Input Data Properties**.
3. In the **Input Data Properties** pane, click **...** to the right of the **Connection** text box.
4. Configure the Web service endpoint and your user credentials.
5. Click **OK**. Java Explorer saves your connection settings.

Web Service Emitter Timeout for Axis 1.x

Note: You do not need to specify timeouts with Axis2, as Axis2 handles timeouts itself.

When you generate Web service stub code for large WSDL files, the process of generating the Java source code may take longer than the default timeout specified by the Axis library. In such a case, the following exception is generated:

```
com.segue.jexplorer.datatyp.exception.JExpWsdlException:
Failed to generate Web Service proxy library(webAxisProxy1.jar).:
Failed to create java source from WSDL.:
java.io.IOException:WSDL2Java emitter timed out (this often means the WSDL
at the specific URL is inaccessible)!
at com.seque.jeplorer.runtime.plugin.impl.webservice.axis.JExpAxisProxyGen
.generateProxyLibrary(JExpAxisProxyGen.java:272) at
com.segue.jexplorer.runtime.plugin.impl.webservice.axis.JExpAxisProxyGen
.generateProxy(JExpAxisProxyGen.java:138) at
com.segue.jexplorer.runtime.communication.JExpRuntimeOperation
.generateWebServiceProxy(JExpRuntimeOperation.java:772)
```

You can specify a WSDL2Java emitter timeout in a configuration file called `webservice.properties`. The property in the file must be called `emitter.timeout`. The value must be specified in milliseconds.

The file is located under `<install dir>\Java Explorer\Startup\webservice.properties` and has the following content:

```
# For generating Web Service stub code with axis libraries
# a timeout in milliseconds may be specified for the emitter to
# generate the source files
emitter.timeout=5000
```

Negative Testing

Although extended support for negative testing is not available, you can ignore the expected exceptions.

Animated Runs

Animated runs or *test runs* execute all test methods that are assigned to your test cases, with corresponding input and output data displayed in the **Input Data** and **Output Data** windows as they are submitted/received. Status symbols indicate the success/failure status of method calls after they have run.

Output of animated runs includes several logs and reports that help you to identify problems with tested components.

To configure settings for animated runs go to **Tools** > **System Settings** > **Options**. The following settings are available:

- **Always run Init Test Case**
- **Always run End Test Case**
- **Run all Test Cases during Animated Run**
- **Invoke Delay**, which is the time delay between method calls, in ms

Note: By default, there is a 2-second (2000 ms) delay between the invocation of method calls. This is because Java Explorer is designed to test the functionality of components, but not to overtax them with excessive load. Invocation delay settings can be configured on the **Options** page.

Executing an Animated Run

To execute an animated run:

1. Click **Run Test** on the Workflow bar. The **Animated Run** dialog box displays.
2. *Alternative:* Click **Run** > **Animated Run**.
3. Check the check boxes that correspond to the test cases that are to be executed during the test run:
    - **InitTestCase**
    - **TestCase**
    - **EndTestCase**

    The default selections vary based on your option settings.

    Note: When test cases depend on each other for accessing global objects, ensure that you do not exclude test cases that define objects that are used by other test cases.
4. Check the **Do verifications** check box to specify that verifications that are defined for output parameters should be performed.
5. Click **OK** to begin the animated run.

    If you have specified a different calling count for your test cases than `1`, the test cases are executed the number of times defined in this property.

    If the number of method calls reporting an error exceeds the number of allowed errors defined in the test case properties, the test case is aborted.

    Note: Animated runs can be aborted manually by either hitting `Esc` on your keyboard or by selecting `Stop Run` from the **Run** menu. Animated runs are aborted automatically when a verification with severity `Error, the simulation is aborted` fails.

    All global variables are initialized to their initial values. All random variables are reset.

Viewing Status of Animated Runs

Depending on the number of times a test case is called, which is defined by the `CallCount` property, you can see which iteration is currently executing in the **Test Scenario** window. Each test case and test method has one of the following status icons:

Status Icon	Description
Green checkmark	OK
Blue *V* icon	Verified
Yellow triangle with exclamation mark	Warning
Red *X*	Error

Methods that fail due to thrown exceptions or failed verifications display failed or warning method call icons next to them. If a method call fails you can view a description of the failure by holding your cursor over the method.

Analyzing Results

Once you have completed a test run, you can view the following three result files to assist you in analyzing test results:

Result File	Description
User report	View the user report generated during the last animated run. The report contains statistical information about each test case and each test method call.
Error report	View the error report generated during the last animated run. The report contains a description for every error that occurred.
Animation log	This file contains information about every method that has been called with their parameters and errors that occurred.

Viewing Result Files

Before proceeding with this task you must execute a test.

1. Click **Results** on the Workflow bar to display the **Explore Results** dialog box.
2. Select the result type you want to generate.
3. *Alternative:* You can also select result files from the Java Explorer **Results** menu or by clicking the **Output** tab above the **Loaded Components** pane.

User Report

The user report is based on the Silk Performer virtual user report. This report offers information regarding time elapsed for each test case and test-method call.

User reports are comprised of four sections:

- Test case summary
- Test case details
- Test method summary
- Errors

The **Test case summary** section offers information about the number of executed test cases and the number of errors that occurred during the test run.

The **Test case details** table gives you information about each test case:

- Number of executions per test case
- Execution time of each test case

The **Test method summary** table offers information about each test method that has been called. You receive information about execution times, number of errors that occurred, and when testing Web Services or .NET Remoting, the number of bytes that were transmitted.

The **Errors** table gives you information about each error that occurred during the test run. You receive information regarding when each error occurred, which test methods caused the errors, and short descriptions of the errors.

Error Report

The error report delivers an overview of all errors that occur during test runs. They include details regarding error time, in which test case and which test method each error occurred, severity types, and error messages.

Animation Log

The animation log on the **Output** tab displays real-time information about the methods that were called in the most recent test run. Additional information about passed parameters is also included. When exceptions are encountered, the animation log offers detailed stack traces for the exceptions.

You can access the animation log at any time by clicking the **Output** tab.

Exploring Object Properties

To explore the properties of an object:

1. Click the **Objects** tab below the **Loaded Components** pane.
2. Right click the object and select **Quick Info**. The **Quick Info** dialog box displays the internal state of the selected object.
3. *Optional:* In the **Quick Info** dialog box, you can additionally view the following oroperties of an object:
    - Click **Statics** to display the static fields of the object.
    - Click **None Public** to display the private, package, and protected fields of the object.
    - Click **Base Types** to display the parent classes of the object.

Exploring Method Properties

To explore the properties of a method:

Right click the object in the **Loaded Components** pane and select **Properties**. The **Method Properties** dialog box displays the following properties:

- Name
- Input parameter types
- Output parameter type
- Exceptions
- Modifiers

System and Project Settings

Java Explorer offers a number of system and project setting options.

System Settings

System settings are saved in the file `JExplorer.conf` located at `C:\ProgramData\Silk\Silk Performer 20.5`.

Accessing System Settings

1. Click **Tools** > **System Settings**. The **System Settings** dialog box opens.
2. Click the tab that corresponds to the options you wish to configure.

   The dialog box provides the following tabs:

   - **Runtime**
   - **Default Values**
   - **Options**
   - **History**
   - **Connection**
   - **JUnit**
   - Silk Performer

Processes

When you start Java Explorer, at least two processes are run. The first process displays the Java Explorer GUI and runs under SDK 1.4. The second process is the runtime process. This process instantiates and manages all Java objects that belong to the currently open project. The runtime process can run under any JDK. You can configure the JDK in the **Project Settings** dialog box.

The runtime process is started with the JDK that is configured and is initialized with the static classpath entries. After the runtime process is initialized, the *UrlClassLoader* is initialized with the dynamic classpath entries.

Configuring Default Value Settings

You can change the input value parameters for method calls. You can set default values for the following value types:

- **Default boolean**, which can either be `true` or `false`.
- **Default double**, which stands for all floating-point numbers.
- **Default int**, which stands for all integer numbers.
- **Default string**.

1. Click **Tools** > **System Settings**. The **System Settings** dialog box opens.
2. Click the **Default Values** tab.
3. Click on the value that you wish to change and type the new value.
4. Click **OK** to confirm your changes.

Configuring Verification Settings

1. Click **Tools** > **System Settings**. The **System Settings** dialog box opens.
2. Click the **Options** tab.
3. In the **Verifications** section, check the check boxes that correspond to the settings that you want to activate.

   The following options define the behavior of verifications:

Option	Description
Automatically define a verification for stored values	If checked, the default verifications are defined whenever you store an output value in a variable.
Ignore Case (default value)	If checked, the verification of string values in the **Verification Wizard** is case insensitive. By checking or unchecking this check box, you also check or uncheck the **Ignore Case** check box in the **Verification Wizard**.
Check verifications during animated runs	If checked, the verification is activated during animated runs. By checking or unchecking this check box, you also check or uncheck the **Do Verifications** check box on the **Animated Runs** dialog box.

4. Click **OK** to confirm your changes.

Configuring Code Generation Settings

1. Click **Tools** > **System Settings**. The **System Settings** dialog box opens.
2. Click the **Options** tab.
3. In the **Code Generation** section, check the check boxes that correspond to the settings that you want to activate.

   The following options define how code is generated when you export to a standalone application, JUnit, or a Silk Performer Java Framework project:

Option	Description
Try-Catch block in every method	By checking or unchecking this check box, you also check or uncheck the init test case in the **Verification Wizard**.
Verifications	If checked, adds verifications to the generated code.
Silk Performer Measures	If checked, adds Silk Performer `MeasureStart` and `MeasureStop` functions to the generated Silk Performer Java Framework code. For additional information about measures, refer to Silk Performer Help.

4. Click **OK** to confirm your changes.

Configuring Animated Run Settings

1. Click **Tools** > **System Settings**. The **System Settings** dialog box opens.
2. Click the **Options** tab.
3. In the **Animated Run** section, check the check boxes that correspond to the settings that you want to activate.

   The following options define the behavior of animated runs:

Option	Description
Always run init test case	By checking or unchecking this check box, you also check or uncheck the init test case on the **Animated Run** dialog box.
Always run end test case	By checking or unchecking this check box, you also check or uncheck the end test case on the **Animated Run** dialog box.
Run all test cases during animated run	By checking or unchecking this check box, you also check or uncheck all test cases on the **Animated Run** dialog box.

4. To define the delay in milliseconds between the invocation of method calls in an animated run, type the delay into the **Invoke Delay** text box.
5. Click **OK** to confirm your changes.

Deleting Loaded-File Entries in History

Java Explorer tracks all classes, archives, and WSDL URLs that are loaded. Instead of typing the full name, you can select an entry from the history list if the class is loaded.

To delete loaded-file entries:

1. Click **Tools** > **System Settings**. The **System Settings** dialog box opens.
2. Click the **History** tab.
3. Select the file that you want to delete in the **Loaded file entries** list box.
4. Click **Delete**.
5. Click **Yes** to confirm the deletion of the file entry.

Connection Settings

Tools > **System Settings** > **Connection**

The **Connection** dialog box enables you to define the default proxy connection that is used to communicate with a Web service. The specified connection is also used to load a WSDL file from the Internet.

When you load a WSDL file, the Web service class inherits the settings in the **Connection** dialog box. If you change the settings in the dialog box after the Web service class is loaded, the changes are not reflected in the Web service class.

JUnit Settings

Tools > **System Settings** > **JUnit**

The **JUnit** dialog box enables you to select the TestRunner that is used to run JUnit tests from the **Kind of TestRunner to use** list box. In the **Archives necessary to run JUnit** section, you can click the **Add archive(s) to classpath** button to specify the classpath information that is required to run the selected TestRunner.

Silk Performer Settings

Tools > **System Settings** > Silk Performer

You can configure the home directory and the project directory for Silk Performer with the Silk Performer dialog box. The initial settings are detected by the Java Explorer setup.

Project Settings

This section describes the project setting options that Java Explorer provides.

Configuring Runtime Settings

To enable a specific JDK version for a project:

1. Click **Tools** > **System Settings**. The **System Settings** dialog box opens.
2. Click **Add** next to the **Target JDK** list box and browse for the Java Home Directory.

   For example, the Sun JDK version 1.4 might be installed under `C:\j2sdk1.4.2_04`. The IBM JDK version might be installed under `C:\WebSphere\AppServer\java`.
3. Click **OK**. You can now select the added JDK version from any project.
4. Click **Tools** > **Project Settings**. The **Runtime** tab of the **Project Settings** dialog box opens.
5. Select the JDK from the **Assigned Runtime** list box.
6. Click **OK**. The Java Runtime is now assigned to the current project.

Note: When you change the active JDK, the runtime process is restarted. Perform an animated run to update the objects in the runtime.

Configuring Web Service Plug-In Settings

Java Explorer supports Axis 1.4, Axis2 1.4 or later, and GlassFish Metro 1.5 out of the box.

Note: Use Apache Axis 1.4 or later to test secure Web services.

Note: Metro does not work with JDK 1.6 unless you install the latest JDK 1.6 updates.

To select a specific SOAP stack version for the project:

1. Click **Tools** > **Project Settings**. The **Runtime** dialog box opens, and you can assign the SOAP stack version that is used for the current project.
2. Click the **Web Service Plugin** tab.
3. Select a SOAP stack library version from the **Assigned Web Service Plugin** list box and click **OK**. The **Properties** list box displays the classpath configuration that belongs to the selected SOAP stack version.
4. Restart the Java Explorer runtime process to change the selected SOAP stack version.
5. Click **Yes** to confirm the restart.

Configuring Classpath Settings

To configure the classpath settings:

1. Click **Tools** > **Project Settings**.
2. Click the **Classpath** tab.
3. Add static classpaths to or remove static classpaths from the **Static** list.

   The static classpaths include entries that are defined as static or system classpaths. These entries are set when the runtime is created. When you change a static or system class you need to restart the runtime.
4. Add classpaths to or remove classpaths from the **Jdk** list.

   The entries in the list reflect the classpaths for the currently configured JDK.

5. Add classpaths to or remove classpaths from the **Dynamic** list.

   The entries in the list are dynamically added to the UrlClassLoader in the runtime process. When you add an entry to the list or change an entry, you do not need to restart the runtime.
6. Check the **Restart Runtime Process** check box to restart the Java Explorer runtime process.

   The check box fulfills two purposes. Firstly, it signals if changes made to the classpaths require the restart of the runtime process, and secondly it allows you to explicitly force a restart of the runtime process.
7. Check the **Show Full Classpath** check box to display all the internally used classpath entries.

   By default, only the static, dynamic, and JDK classpath entries are displayed and can be customized. When you check the check box, all other classpath entries, for example the Java Explorer archives, the Web service plug-ins, and others, are also visible. This is helpful for expert users and certain internal purposes, for example if you want to see if an archive of a test of a customer environment conflicts with a Web service plug-in.

   Note: The classpath tree is ordered by priority. Archives at the top of the tree have priority over archives at the end. You can move the entries up and down within a group or even between neighboring groups, unless the groups are read-only.

   Note: Check boxes indicate whether a classpath is static or dynamic, which means whether a change to the classpath group requires a restart of the runtime process or not.
8. *Optional:* You can also edit runtime-specific classpath entries through the **Project Settings** dialog box.

   Note: The runtime classpath belongs to the System Settings and changes to it effect all projects. The classpath is stored in both the `JExplorer.conf` system settings file and the JEP project file. When you load a project, if the classpath entries in the project file do not match the classpath settings in the system settings, the system settings override the stored project settings.

Configuring Scripting Settings

The **Scripting** dialog box defines the file names that are used to export the Java Explorer project.

To configure the scripting settings:

1. Click **Tools** > **Project Settings**.
2. Click the **Scripting** tab.
3. Type the file name of the exported Java test class for the Silk Performer Java Framework into the **TestClass for** Silk Performer text box.
4. Type the file name of the exported JUnit test class into the **TestClass for JUnit** text box.

   This class extends `junit.framework.TestCase`.
5. Type the file name of the exported standalone test class into the **TestClass for Standalone** text box.

   You can execute the class from the command line or integrate it into other test frameworks.
6. Type the name of the BDL script file, that is generated for the Silk Performer Java Framework, into the **TestScript for** Silk Performer text box.

Exporting Projects

Once you have defined your test scenario you can either continue running tests in Java Explorer or you can export the underlying test code to the following environments:

Environment	Description
Silk Performer Java Framework Project	Export to Silk Performer to run tests based on the scenario you have defined in Java Explorer.
JUnit Test Case	Export to JUnit to directly execute the test cases with the JUnit Run Test Suite.
Standalone Console Application	Run the test scenario as a standalone application, independent of Java Explorer, or use the test scenario to record Web traffic with Silk Performer.

Exporting a Java Explorer Project

To export a Java Explorer project:

1. Click **Export**.
2. Click on one of the following buttons to select the environment to which you want to export the test scenario:
   - **Java Project** Silk Performer
   - **JUnit Test Case**
   - **Standalone Console Application**
3. Specify a name for the exported file in the corresponding text box.
4. Specify a destination for the exported file in the **Directory** text box.

   By default, Java Explorer saves projects to the project directory at `<my documents>\Silk Performer 20.5\Projects`.

   Note: If you are calling a Web service in the Internet, ensure that you have set the correct proxy in **Tools** > **Project Settings** > **Classpath**.

Silk Performer Java Projects

You can create Silk Performer Java projects with the Java class and a BDF file that contains the calls to the Java class. The *Java Profile Settings* are saved in the Silk Performer project file, so the same Java Runtime and classpath settings are used in Java Explorer and Silk Performer. You can compile the Java test class in the Silk Performer GUI. When you start a test or a try-script run, the Java compiler and the BDL compiler are automatically invoked.

Java Explorer exports the following files to Silk Performer:

- The Silk Performer project file, which is of type LTP.
- The Silk Performer script file, which is of type BDF.
- The Java test script, which is of type JAVA.

Silk Performer Java Framework

The Silk Performer Java Framework enables developers and QA personnel to coordinate their development and testing efforts while working within their specialized environments. The developers work exclusively in their Java-programming environment while the QA personnel work exclusively in Silk Performer. The Silk Performer Java Framework thereby encourages efficiency and a tighter integration of QA and development.

Exporting to Silk Performer is valuable for QA personnel who are proficient in facilitating analysis of Java components with Silk Performer and wish to create Silk Performer projects, test case specifications, and scripts from which they can run tests.

When you directly export a project to Silk Performer as a Java project, a default Silk Performer project file of type LTP is created. You can then test a project with multiple virtual users. You can also gather and analyze performance data for each method call in the project.

Working with Multiple Agents

Usually, load testing with Silk Performer is done through multiple agents. To enable the framework to work smoothly:

1. Install a JDK on each agent machine.

   The version of the JDK should be the same version, or later, as the one used for the Java Explorer project.

   Note: Install the JDK in the same directory on each machine.
2. Ensure that the same classpath is available on each agent machine.

   For example, to load test a WebLogic EJB, you must install the package `weblogic.jar` in the same directory on each agent machine. The same applies to EJB stubs.

JUnit Test Cases

JUnit is a widely used regression testing framework. You can export JUnit TestSuites with Java Explorer. The export generates a Java test class for JUnit and two batch files for compiling and running the test class. For additional details on JUnit, visit *https://bugs.eclipse.org/bugs/*.

Exporting a JUnit TestSuite

To export a JUnit TestSuite:

1. Click **Export** > **JUnit Test Case**.
2. Click **Open Dialog ...** next to the **File Name** text box. The **Scripting** page of the **Project Settings** dialog box opens.
3. Specify a name for the exported file in the **Test Class for JUnit** text box.
4. Click **OK**. The **Project Settings** dialog box closes.
5. Click **Browse ...** next to the **Directory** text box to select the target directory.
6. Check the **Open Test Suite in JUnit** check box to start the JUnit testrunner directly from the Java Explorer GUI.
7. Click **OK**. Java Explorer confirms that the settings are correct. If the settings are not correct, the **JUnit** page of the **System Settings** dialog box opens. You can select the TestRunner that is to be used to run JUnit tests from the **Kind of TestRunner to use** list box. The **Add archive(s) to classpath** button next to the **Archives necessary to run JUnit** box enables you to specify the classpath information required to run the selected TestRunner.

Test Case Execution Sequence

There are differences in the way that JUnit and Java Explorer run tests. In JUnit TestSuites, values and global variables that need to be initialized in the `init()` test case cannot be passed from one test case to the next.

Example

The execution sequence of the following Java Explorer project, that contains an `init()` test case, an `end()` test case, and two test cases, is:

1. InitTestCase
2. TestCase1
3. TestCase2
4. EndTestCase

In JUnit, the execution sequence of the same project is:

1. InitTestCase (setUp)
2. TestCase1 (testMethod1)
3. EndTestCase (tearDown)
4. InitTestCase (setUp)
5. TestCase2 (testMethod2)
6. EndTestCase (tearDown)

Standalone Console Applications

With Java Explorer, you can compile the test cases in a test scenario into a *standalone console application*, which is a Java class that can run independent of Java Explorer, for example from the commandline. A standalone console application calls all test methods in the main method.

You can use standalone console applications to either run test scenarios independent of Java Explorer, or to generate network traffic that can be recorded by Silk Performer. Silk Performer can generate Web BDL scripts based on such network traffic.

Exporting Standalone Console Applications

1. Click **Export** > **Standalone Console Application**. The **Export Standalone Java application** dialog box opens.
2. Type a name for the exported standalone console application into the **File name** text box.
3. Click **Browse ...** next to the **Directory** text box and select the destination directory for the standalone console application.
4. Click **OK**. A JAVA and a BAT file are saved to the current project directory by default.
5. Open a command prompt in the folder to which you have exported the standalone console application.
6. Run `<project name>.bat`, where <*project name*> is the name of your current Java Explorer project.
7. Check the status information which is listed in the command window.

Recording a Standalone Console Application

To record a standalone application with Silk Performer:

1. Launch Silk Performer.
2. Click **File** > **New Project** and create a new Web project.
3. Create a new recording profile for the standalone console application.
4. Specify a name for the profile.
5. Specify the path to the batch file.
6. Specify `Custom Application` as the application type.

7. Select **Winsock**.
8. Begin recording the application. After recording you should have a Web BDL script that contains all the HTTP/TCP calls of the console application to a Web service.

Index